Mobile Streams: A Middleware for Reconfigurable Distributed Scripting

M.Ranganathan, Virgnie Schaal, Virginie Galtier and Doug Montgomery
Internetworking Technologies Group
National Institute of Standards and Technology
100 Bureau Drive, Gaithersburg, MD 20899.
{mranga, schaal, vgaltier, dougm}@antd.nist.gov

Abstract

A large class of distributed applications follow an event-driven or reactive paradigm. Such applications can benefit from Mobile Agent technology by making it easy to add re-configurablity, extensibility, and failure resilience features at an application level. We present the design of a Middleware for building reactive, extensible, reconfigurable distributed systems, based upon an abstraction we call Mobile Streams. Using our system, a distributed, event-driven application can be scripted from a single point of control and dynamically extended and re-configured while it is in execution. Our system is suitable for building a wide variety of applications; for example, distributed test, conferencing and control-oriented applications. We illustrate the use of our system by presenting example applications.

1 Introduction

Current trends indicate that in the future, the architecture of distributed systems will be radically different from those of today. The forces propelling such changes include: (1) the movement towards making embedded processing, sensors and actuators first order components of the networked computational infrastructure; (2) the need to accommodate environments in which semi-autonomous systems and devices organize into cooperating systems and (3) the need to enable distributed control systems to dynamically adapt and optimize their behavior in reaction to changing environments and physical composition of the system components.

Consider a Middleware framework that allows the components of a distributed application to be moved around from machine to machine dynamically (i.e. topological re-configuration) without stopping. There are classes of applications where a dynamic re-configuration of the distributed system can significantly enhance system performance. For example, envision a control application consisting of *reactive* distributed components. Such an application implements a distributed on-line algorithm, adjusting system parameters in response to sensor inputs. In order to achieve good control performance and achieve system stability, it is necessary to situate the control logic close to the process being controlled. However, it may not be possible to know a-priori what constitutes a good placement. The distributed control application can react to changes in the environment and resources by dynamic placement and movement of control elements. Such re-configurations may be made from outside of the distributed system by a global observation of the environment (*global reconfiguration*) or from within the application itself (*autonomous reconfiguration*).

Next, consider a capability that permits dynamic updating of functionality (code) in a distributed application. Certain classes of long-running distributed applications can benefit from such a capability. For example, consider a distributed system where nodes (sites) may be dynamically added and removed. In reaction to this, the code controlling a distributed process running on these nodes may need to be updated. However, it may not be necessary or allowable to stop the system in order to achieve this. In this situation, having the ability to dynamically update functionality without shutting down the system becomes a requirement.

The examples above describe some of the considerations that motivate the design of the Middleware framework that we present in this paper. Our approach is based upon an abstraction we call *Mobile Streams (MStreams)*. The significant attributes of our approach include:

1. **Dynamic re-configurablity and extensibility:** Our Middleware allows for multiple points of control; that is, the distributed application can be reconfigured and extended by initiating actions from any of the participating sites. Such applications may be dynamically extended and reconfigured without stopping the system or flushing messages.
2. **Support for low-latency, distributed reconfiguration:** We use distributed lazy caching and multi-threading to reduce the cost of topological reconfiguration.

3. **Peer-to-peer messaging:** In order for a distributed application to continue to operate while it is being re-configured, we need a peer-to-peer communication protocol that preserves ordering while these actions are taking place. We have developed an efficient peer-to-peer asynchronous communication protocol that preserves ordering and reliable delivery in the presence of failures and dynamic reconfiguration. Ordered, reliable messages greatly simplify the design of distributed applications.

The rest of this paper is organized as follows: Section 2 presents the *MStreams* programming model and system architecture and presents an introductory example. Section 3 gives a brief overview of *AGNI*, our prototype implementation of *MStreams* Middleware and presents some preliminary performance results. Section 4 presents some more comprehensive applications that we have built on our system. In Section 5 we compare and contrast our work with those of others. In Section 6 we conclude and present our future plans for this project.

2 Mobile Streams

A Mobile Stream (*MStream*) is a named communication end-point in a distributed system that can be moved from machine to machine while a distributed computation is in progress and while maintaining a pre-defined ordering guarantee of message consumption with respect to the order in which messages are sent to it.

An *MStream* has a globally unique name. We refer to any processor that supports an *MStream* execution environment as a *Site*. The closest analogy to an *MStream* is a mobile active mailbox. As in a mailbox, an *MStream* has a globally unique name. *MStreams* provide a *FIFO* ordering guarantee, ensuring that messages are consumed at the *MStream* in the same order as they are sent to it. Usually mailboxes are stationary. *MStreams*, on the other hand, have the ability to move from *Site* to *Site* dynamically. Usually mailboxes are passive. In contrast, message arrival at an *MStream* potentially triggers the concurrent execution of message consumption event handlers (*Append Handlers*) registered with the *MStream*, which can process the message and, in turn, send (*append*) messages to other *MStreams*.

A distributed system consists of one or more *Sites*. A collection of *Site*s participating a distributed application is called a *Session*. Each *Session* has a distinguished, trusted, reliable *Site* called a *Session Leader*. Each *Site* is assigned a *Location Identifier* that uniquely identifies it within a given *Session*. New *Sites* may be added and removed from the *Session* at any time. An *MStream* may be located on, or moved to any *Site* in the *Session* that allows it to reside there. *MStreams* may be opened like sockets and messages (*appended*) to them. Multiple Event Handlers (*Handlers*) may be dynamically attached, to and detached from, an *MStream*. *Handlers* are invoked on discrete changes in system state such as message delivery (*append*), *MStream* relocations, new *Handler* attachments new *Site* additions and *Site* failures. We refer to these discrete changes in system state as *Events*. *Handlers* are attached by *Agents* which provide an execution environment and thread for the *Handlers* that they attach. (i.e. an *Agent* specifies a collection of *Handlers* that that all use the same thread of execution and interpreter.) Logically, the system is structured as shown in Figure 1.

Handlers can communicate with each other by appending messages to *MStreams*. These messages are delivered asynchronously to the registered *Append Handlers* in the same order that they were issued [1]. A message is *delivered* at an *MStream* when the *Append Handlers* of the *MStream* has been activated for execution as a result of the message. A message is *consumed* when all the *Append* handlers of the MStream that are activated as a result of its delivery have completed execution. By *asynchronous delivery* we mean that the sender does not block until the message has been consumed in order to continue its execution.

Our architectural goal is separation of logical design of a distributed application and physical placement of its components. A distributed application is constructed by first specifying the communication end-points as *MStreams* and then attaching *Agents* to those end-points, that in turn attach *Handlers* for specific *Events*. A given *MStream* may have multiple *Agents* and each *Agent* may register *Handlers* for different *Events*, but each *Agent* may have only one *Handler* for a given *Event*. When an *Event* occurs, the appropriate *Handler*s in each *Agent* are concurrently and independently invoked with appropriate arguments. *Handlers* are typically registered on *Agent* initialization and may be dynamically changed during execution.

An application built using our Middleware, may be thought of as consisting of two distinct parts - an active part and a reactive part. The reactive part consists of *Streams* and *Handlers*. The active part or *Shell* lives out-

[1] Synchronous delivery of messages is supported as an option but asynchronous delivery is expected to be the common case.

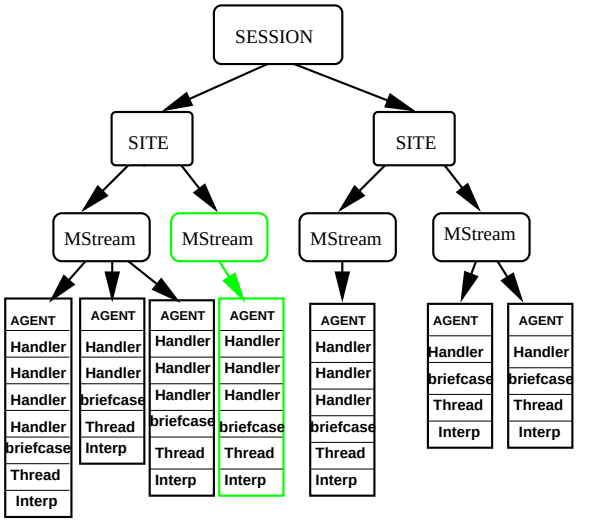


Figure 1. Logical organization of the System. A Session consists of multiple participating Sites. Each Site can house multiple MStreams. Each MStream can have multiple Agents that can register Handlers for different Events. MStreams can move from Site to Site. When an MStream moves, all its registered handlers move with it.

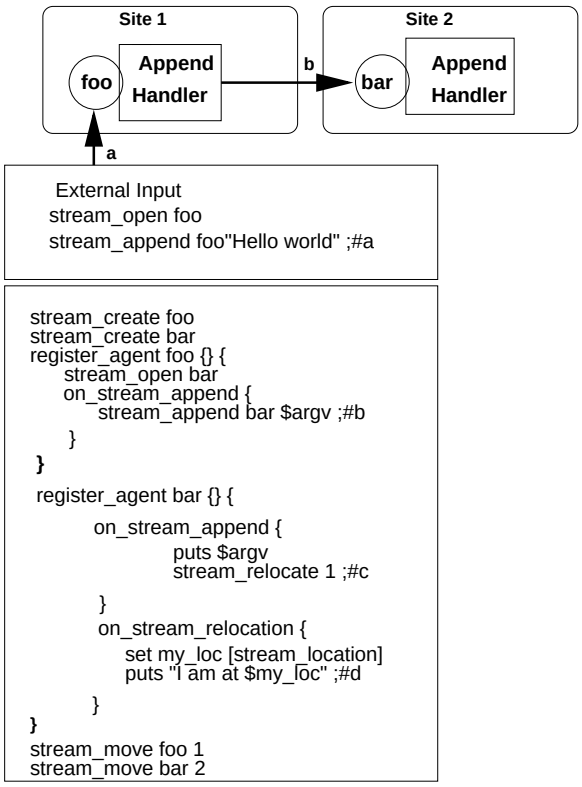


Figure 2. A simple auto-reconfiguring reactive system scripted from a single point of control.

side the Middleware and drives it. A *Shell* may connect to the Middleware and issue requests and may exit at any time. The reactive part is persistent.

Figure 2 shows an example script that instantiates a simple distributed system that resides at *Sites* 1 and 2. A message is sent to the stream called *foo* by the `stream_append` command issued via the external *Shell* (#a in Figure 2) . The *MStream* called *foo* receives the message "Hello world" and sends it to the *MStream* called *bar* (#b in Figure 2) which outputs the message via its handler and then moves *MStream bar* to *Site* 1 (#c in Figure 2). The arrival handlers run when the *MStream bar* arrives at *Site* 1, printing the string "I am at 1" to the console at Site 1 (#d in Figure 2).

In Figure 2 the script labelled "External Input" in is the *Shell* and *MStreams* and their registered handlers are the reactive parts.

2.1 Dynamic Extension and Re-configuration

An application built on our Middleware may be dynamically extended and re-configured in several ways while it is in execution (i.e., while there are pending un-delivered messages). First, an *Agent* can dynamically change the handlers it has registered for a given *Event*. Second, new *Agents* may be added and existing *Agents* removed for an existing *MStream*. Third, new *MStreams* may be added and removed. Fourth, new *Sites* may be added and removed, and finally, *MStreams* may be moved dynamically from *Site* to *Site*.

When an *MStream* moves from one *Site* to another, it (logically) moves the code of all of the *Agents* attached to it to the new *Site* along with whatever state they have placed in their *briefcase* structures. We say an *Agent* "visits" a *Site* when its *MStream* visits the *Site*. When an *Agent* first visits a *Site*, its initialization code executes there and when an *Agent* is killed, its (optional) *Finalization Handler* runs at each location that has been visited by the *Agent*. *Agent* state (consisting of global state variables and code) is replicated at each site that it visits until the *Agent* is destroyed. On *Agent* destruction, the *Handlers* that it has registered are de-registered, and the interpreter and state variables are freed at each *Site* that it has visited. We assume that *Sites* may fail or disconnect during execution. *Site* failure does not imply destruction of the *MStreams* that reside there. Failure processing is

described in Section 2.4.

The *Agent's briefcase* specifies a consistency requirement for moves. When an *Agent* moves from *Site* to *Site* only the elements in the *briefcase* are copied from the source execution environment to the target. The remainder of the global state remains unaffected (and cached) at the source site of the move. On successful completion of a move, the *Arrival Handler*s of the *MStream* are invoked at the new *Site* where the *MStream* has moved.

Handlers may move the *MStream* to which they are attached and also may move other *MStreams* around as well as create and destroy *MStreams*. *Handlers* may also exit - destroying the *Agent* in which they are housed and may also destroy other *Agents* . Such actions may also be initiated from an external *Shell*. Re-configuration may be contained by using appropriate policy handlers, as described in section 2.2.

All changes in the configuration of an *MStream* such as *MStream* movement, new *Agent* addition and deletion, and *MStream* destruction are deferred until the time when no *Handler*s of the *MStream* are executing. We call this the *Atomic Handler Execution Model* . Message delivery order is preserved despite dynamic reconfiguration, allowing both the sender and receiver to be in motion while asynchronous messages are pending delivery.

2.2 Restricting Extension and Re-configuration

Applications built using *Mobile Streams* can be extended from multiple points of control; any handler or *Shell* (see Section 2) that has acquired an open *MStream* handle, can attempt to re-configure or extend the reactive part of the system and these actions can occur concurrently. While this adds great flexibility, it also raises several security and stability issues. We provide a means of restricting system reconfiguration and extension using control *Event*s that can invoke policy *Handlers*. These policy *Handlers* may be registered only by privileged *Agents* as described below. We follow a discretionary control philosophy by providing just the mechanism and leaving the policy up to individual applications. Controls may be placed via policy *Handlers* at a session-wide level, site-wide level and at the level of individual *MStreams* for various security-relevant *Events*.

The *Session Leader MStream* is a trusted, stationary *MStream* that resides in the distinguished *Session Leader Site* (see section 2) and may have a single stationary *Agent* that can register *Handlers* for *Session Control Events*. Using this mechanism, controls may be placed on various system-wide events such as new *MStream* creation, *MStream* destruction, new *Agent* creation and destruction and *MStream* motion.

Each *Site* has a single stationary *Site Controller MStream* with a stationary *Agent* resident at that site that can register *Handlers* for *Site*-specific *Event*s such as *MStream* open, *MStream* arrival, new *Agent* registration and *Shell* connection and disconnection *Event*s.

Before an *MStream* is accepted at a given *Site*, the *MStream arrival* policy *Handler* responsible for accepting or denying the *MStream* entry can query various properties about its identity and registered handlers. The *Site Controller* may also specify a code fragments to be executed in the context of any new *Agent* that gets created or initialized at its *Site*. *Site*-specific code can thus be made to intervene in security-sensitive operations such as file opens. This provides a means of "sandboxing" *Handlers* that execute at a *Site*. Using the mechanisms offered by safe TCL, for example, a function may be constructed that intervenes in sensitive operations such as channel opens.

At the time of its creation, each *MStream* may specify a privileged *Stream Controller Agent*. The *Stream Controller* remains associated with the *MStream* for its lifetime and may not be destroyed once it is created without destroying the *MStream* itself. The *Stream Controller* can register privileged *Handlers* for various *Events*. *Handlers* registered by other (non *Stream Controller*) *Agent*s are considered "non-privileged". The set of *Events* for which a non-privileged *Agent* may register *Handlers* is a proper sub-set of the set of *Events* for which the *Stream Controller* may register *Handlers*. The *Stream Controller* can place policy *Handlers* that can intervene in various sensitive operations such as new *Agent* registrations, *MStream* opens and *MStream* movement, in addition to being able to register *Handlers* for message consumption (*append*) *Events*, relocations and failures. There can be at most one privileged *Handler* for a given *Event*. For any *Event* for which both non-privileged *Handler*s and a privileged *Handler* exists for a a given *MStream* , the privileged *Handler* gets control first and has to activate the non-privileged handlers for execution. Hence, for a given *Event*, the execution of a non-privileged *Handler* may be controlled by the corresponding privileged *Handler*. This mechanism is useful in constructing distributed debuggers and for enforcing *MStream*-specific policies.

As an example of how these security mechanisms operate, Figure 3 shows the different policy checks that can be made during *MStream* Open. A *stream_open* request may originate from a *Handler* or external *Shell* (#a in the figure). If the caller does not already have an open handle, the request is forwarded to the *Session Leader* where it is vetted by the *Session Leader* reg-

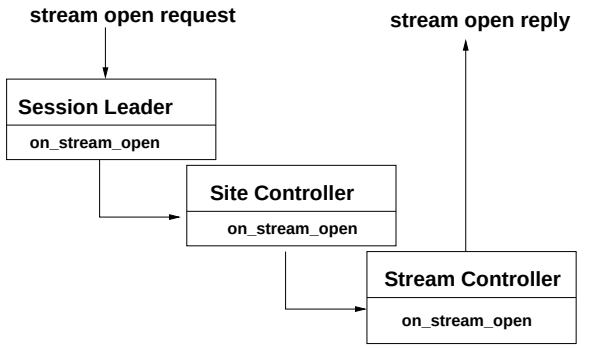


Figure 3. Checking of actions may be made at several levels - Session-wide, Site-specific and Stream-specific by attaching policy Handlers that intervene in these actions.

istered *on_stream_open* policy *Handler*. If this check passes, the request is forwarded to the Site where the *MStream* resides where the request is vetted by the *Site Controller* registered *on_stream_open* policy *Handler*. If this check passes, the request is forwarded to the *on_stream_open Handler* registered by the *Stream Controller* where it may again be accepted or denied. The request may be denied at any level and failures may occur while the request is being processed. If a failure or denial should occur at any step of the request processing, an error is returned to the *Handler* or *Shell* that initiated the request.

The mechanisms described above permit us to build highly flexible and extensible distributed reactive systems that are able to extend and re-configure themselves, and also to place constraints on how the system can be re-configured and extended. Our overall approach for imposing these restrictions differs from other architectures [5, 9, 13] while building on some ideas adopted by those systems. For example, the *Site Controller* approach and sandboxing is followed by other systems. We have added two innovations. First, we have built mechanisms to place session-wide controls over extension and reconfiguration via the *Session Leader*. Second, the *MStream* itself is regarded as an extensible entity to which *Agents* can be attached and detached. It can carry its own policy *Handlers* to allow or disallow such actions, as determined by its *Stream Controller Agent*.

2.3 Message Delivery

Within our Middleware framework, point-to-point messages are delivered using an in-order sender-reliable delivery scheme built on top of UDP. All messages are consumed in the order they are issued by the sender despite failures and reconfigurations. These ordering and delivery guarantees make it simpler to design distributed systems.

In our scheme, the sender of the message is responsible for re-transmitting the message on timeout. We use a sliding-window acknowledgement mechanism similar to those employed by TCP. The sending *Site* buffers the message and computes a smoothed estimate of the expected round-trip time for the acknowledgment to arrive from the receiver. If the acknowledgment does not arrive in the expected time, the sender re-transmits the message. The sender keeps a window of unacknowledged messages and controls flow by dynamically adjusting the width of this window depending upon whether an ACK was received in the expected time or not. Thus far, our description is similar to the mechanisms employed by TCP. We have implemented our own protocol, rather than just use TCP, because TCP does not address certain conditions such as failures above the transport level and dynamic movement of the communicating end-points.

As previously described, an application can be dynamically reconfigured at any time with both the sender and receiver moving. When movement of an *MStream* occurs, a *Location Manager* is informed of the new *Site* location where the *MStream* will reside. This information needs to be propagated to each *Handler* or *Shell* that has opened the *MStream*.

When the target of an *append* moves, messages that have not been consumed have to be delivered to the *MStream* at the new *Site*. There are two design options in dealing with this problem - either forward unconsumed messages from the old *Site* to the new *Site* or re-deliver from the sender to the new *Site*. Forwarding messages has some negative implications for reliability. If the *Site* from which the *MStream* is migrating dies before buffered messages have been forwarded to the new *Site*, these messages will be lost. Hence, we opted for a sender-initiated retransmission scheme. The sender buffers the message until it receives notification that the handler has run and the message has been consumed, re-transmitting the message on time-out.

When an *MStream* moves it takes various state information along with it. Clearly, there is an implicit movement of handler code and Agent execution state (via the briefcase), but in addition, the *MStream* takes a state vector of sequence numbers. There is a slot in this vector for each "alive" *MStream* that the *MStream* in motion has sent messages to or received messages from. Each slot contains a sent-received pair of integers indicating

the next sequence number to be sent or received from a given *MStream*. This allows the messaging code to determine how to stamp the next outgoing message or what sequence number should be consumed next from a given sending *MStream*.

2.4 Handling Failures

A failure occurs when the *Site* where the *MStream* resides fails or disconnects from the *Session Leader*. Each *MStream* is assigned a reliable *Failure Manager Site*. When a such a failure occurs each of the *MStream*s located at the *Site* that has failed are implicitly relocated to its *Failure Manager Site* where its *Failure Handler*s are invoked. Failures may occur and be handled at any time - including during system configuration and reconfiguration. Pending messages are delivered in order, despite failures. A message is considered "consumed" only after all of the *append* handlers execute at the target *MStream* for that message. (If none exist the message is discarded at the recipient). If the *Site* housing an *MStream* should fail or disconnect while a message is being consumed or while there are messages that have been buffered and not yet delivered, re-delivery is attempted at the *MStream Failure Manager*. To ensure in-order delivery in the presence of failures, the message is discarded at the sender only after the *Append Handler*s at the receiver have completed execution and the ACK for the message has been received by the sender. This is different from TCP where the receiver ACKs the message immediately after reception (and not after consumption as we require). After a failure has occurred at the site where an MStream resides, a failure recovery protocol is executed that re-synchronizes sequence numbers between communicating *MStreams* that involve the failed *MStream.* Each of the potential senders is queried to obtain the next expected sequence number. FIFO ordering can be thus be preserved despite the failure.

3 Implementation

We have implemented the *Mobile Streams* model in a toolkit we call AGNI [2] . AGNI is a multi-threaded TCL extension that uses the thread-safety features of TCL 8.1 and consists of roughly 23,000 lines of C++ code. Our system currently runs on Solaris, Linux and Windows NT and may be downloaded from from http://www.antd.nist.gov/itg/agni/. In this section, we give highlights of the implementation some initial performance results. Experimental results were obtained on 150 MHz Ultra SPARC workstations using an unloaded 10 MBPS Ethernet connecting the workstations and simulating packet loss by dropping packets at the receiver.

Each workstation that wishes to participate in the distributed system runs a copy of an *Agent Daemon*. A distinguished *Agent Daemon* houses the *Session Leader* and is in charge of accepting or rejecting new *Agent Daemons*. This *Daemon* also serves as a *Location Manager* and *Failure Manager* for all *MStreams* in the *Session*. Each *Agent Daemon* has a unique identifier that it obtains from the *Session Leader*. Each *Agent Daemon* maintains a connection with the *Session Leader Agent Daemon*. Conceptually, the arrangement is as shown in the figure 4.

Each *Agent* has a TCL interpreter and thread of execution that is used by the *Handler*s that it registers. These resources are created for an *Agent* at a *Site* on its the first visit to the *Site* and remains allocated until the *Agent* (or the *MStream* to which it is attached) is destroyed. When a new *Agent* is added to an *MStream*, its code is propagated and initialized on the first move of the *Agent* to a previously unvisited *Site*, and remains cached there until it is destroyed. Provided an *MStream* has visited a *Site* previously, and no new *Agents* have been attached since its last visit, *MStream* movement simply consists of moving the state information in the *briefcase* (see Section 2) of each *Agent* of the *MStream* to the new *Site* and concurrently invoking each *on_stream_arrival Handler*. For a single handler with a minimal *on_stream_relocation* handler and *briefcase*, moving round robin over 4 locations, our experiments showed a move latency of 38.5 milliseconds averaged over 1000 hops.

Except for the case when the *MStream* is co-located with the *Site* from where the message originates, all control *Events* destined for an *MStream* (e.g. creation, relocation, new agent attachment) are delivered through the *Session Leader Agent Daemon* via the TCP connection that each *Agent Daemon* maintains with it. The *Session Leader Agent Daemon* also acts as a *Location Manager*, keeping track of where each *MStream* is located and is hence able to re-direct control message to the location of the MStream. Sending all control *Events* through the *Session Leader* is a simple means of achieving a global ordering on control messages. The negative aspect of this design is that the *Session Leader* has the potential of becoming a bottleneck. However, we expect the number of control messages to be much smaller than the number of data messages (*appends*) processed by the

[2] "AGents at NIst" (also Sanskrit for fire)

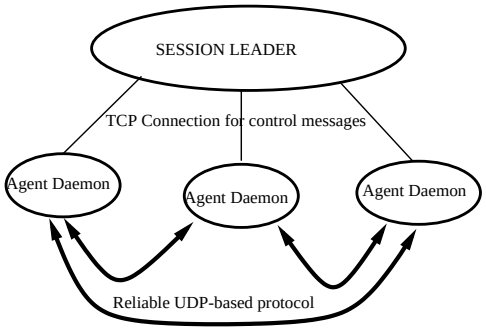


Figure 4. Each Site runs an Agent Daemon that is connected to the Session Leader. The Agent Daemon is Multi-threaded with one thread per agent. The Session Leader maintains location and cache information.

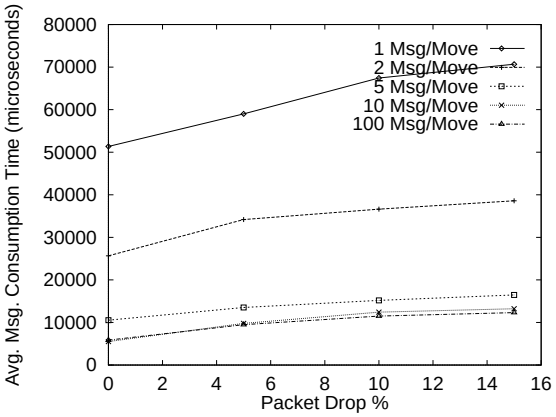


Figure 5. Performance of point-to-point messages for moving endpoints for different message drop percentages. Both sender and receiver are in motion with the sender sending messages to the receiver while in motion.

MStream and hence do not consider this a serious limitation at present. In our future work, we plan to alleviate this problem by replication of the *Session Leader*. The *Session Leader Daemon* also manages the tracking information for the code and state cache described previously and is charge of propagating code to previously unvisited locations. As all code is registered at the *Session Leader* and propagated from there, this simplifies the trust model to pair-wise relationships between each site and the *Session Leader*. provided all parties trust the *Session Leader*.

Appended data messages are delivered to the destination *MStream* directly without going through the *Session Leader*. Thus the *Session Leader* is not a bottleneck for data message delivery.

We performed limited experiments to test the efficiency of our message delivery protocols and system architecture. Figure 5 shows the results of one such experiment. In this experiment, there is one sender *MStream* and one receiver *MStream*. The receiver and sender both move once every K messages. The *Append Handler* at the receiver does little other than reposition the receiver. We assumed a simple congestion model – messages are dropped at the reciever randomly according to a uniform probability distribution. We measured the average time for message consumption as a function of message drop percentages for different values of K. As previously described, our protocol maintains a pipeline of messages between sender and receiver, similar to TCP. On each move, the message pipeline between sender and receiver is broken, and in the limiting case of one move per message, there is at least a round-trip latency and the overhead of communication with the *Session Leader* for each message consumed, thereby resulting is degraded performance. We defer a thorough performance analysis and description of the implementation to a more detailed report.

4 Applications

Several prototype applications have been built using AGNI. In each case, we adopted a problem-driven approach in mapping AGNI capabilities to prototype solutions. For example, we started with the assumption that we would use mobility only to the extent that it simplified the application design in some fashion, rather than adopt the approach that mobility is a feature whose utility needed to be demonstrated. The remainder of this section details the design of two prototype applications.

4.1 Synchronous Collaboration

Synchronous (real-time) collaboration is a mode of computer supported collaborative work where the participants send messages to each other in real-time in order to share a workspace. Common applications that fall under this category include shared white-boards and network chat.

In this section, we present a simple self-reconfiguring network chat application that illustrates how topological reconfiguration may be used to minimize latency in such applications. Consider a network chat application where each participant in the chat can send messages to

all other participants. It may be desired, in such an application, that all participants see an evolving conversation in the same global order. That is, the distributed system requires a globally consistent message consumption ordering.

To simplify our presentation, we assume that there are three fixed participants in the conversation labeled 1,2 and 3 (see Figure 6(a)). Each participant, hosts an *MStream* S_i on her workstation i. The purpose of this *MStream* is to consume messages and display output. Global ordering is achieved by each participant sending messages to a central *MStream* named *M* which re-broadcasts messages to all participants.

In order to guarantee global ordering, all participants must send the messages to *M* and only consume the messages that are re-broadcast from *M*. If *M* were at a fixed *Site*, the latency involved in this operation could become irritating for an interactive user. We solve this problem by employing mobility to periodically re-position *M* to a *Site* that is favorable to the most interactive user. The *Agent* of *M* has a vector counter V with an entry V_i for each user i. When a message is received from a user i, the *Append Handler* of *M* increments the count V_i associated with i. It periodically determines the *Site* from which the most messages originated and moves itself over to that *Site*, zeroing out the vector counter in doing so.

Figure 6(B) shows the Agent code for the central dispatcher M. The *Agent* for M specifies an *on_stream_append* and an *on_stream_relocation* handler and initialization code. The initialization code initializes a counter array (#1) and opens each of the display streams (#2). When a message is delivered to the MStream *M* the dispatchers on_stream_append handler runs. It responds by noting the location from which the message originated (#3) and re-dispatching the message to each of the display *MStreams* (#4). Every 50 messages, it re-positions itself to the *Site* from where the most messages originated (#5), clearing its counters on arrival at the new *Site* (#6).

The remaining pieces of the application consist of the display handler and the input handlers. The input handler (Figure 6.D) runs in an extended TCL shell. It reads input from the keyboard and sends it the the central dispatcher (#2). The Display Handler code is shown in Figure 6(C). It consists of an *MStream* (#1) with an *Append Handler* that simply echoes any message appended to it (#3).

Our performance metric in this application is the expected round-trip time for the "most interactive" user. The effectiveness of this scheme in reducing latency for this user will depend upon several factors - in particular upon network latencies and the expected asymmetry in the frequency of messages sent to the dispatcher by the participants and the expected cost of re-configuration.

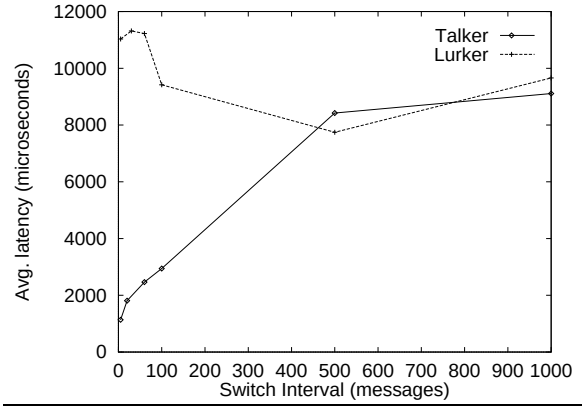
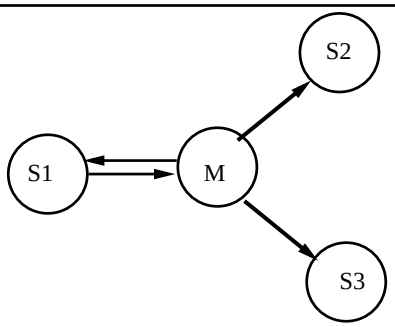


Figure 7. Avg. round-trip time for messages for the adaptive chat example for different dispatcher repositioning intervals. There are 3 participants. Talker role propagates round-robin. See Section 4.1.

We performed a limited experiment to test the efficiency of this scheme using AGNI. In our test, we assumed that there were three participants. At any given time, there is an interactive participant (*Talker*) who generates messages at a frequency of 1 message every 100 milliseconds. The other users (*Lurkers*) generate messages at a rate of one message every 400 milliseconds. Each participant assumes the role of *Talker* in round-robin fashion. We simulated a network packet loss rate of 5% by dropping packets at the receiver.

Figure 7 shows the average round trip time for the messages generated by the *Talker* and the *Lurkers.* The X axis shows the number of messages between moves of the central dispatcher. Clearly, for the *Talker*, it is most advantageous to quickly sense where she is located and move the dispatcher to her workstation. For *Lurkers,* this temporarily results in increased message latency. As the frequency of moves increases, the *Talker* sees improved performance. Our message rate is not high enough to entirely fill the pipeline and hence, in this case, pipeline breakage effects are not immediately apparent and the *Lurker's* performance remains fairly constant for different dispatcher repositioning frequencies. A more detailed set of performance results using such an adaptive algorithm under a system that we had developed earlier is presented in [12].

Based on this principle, we have developed a toolkit



A. Logical Structure of Chat Application

```
set my_id [my_id]                                   ;#1
set mydisplay [stream_create S$my_id ]    ;#2
register_agent $mydisplay "" {
        on_stream_append {
             puts stdout $argv                      ;#3
        }
}
```

C. Display handler

```
set dispatcher [stream_open M]
while { 1 } {
   set input [gets]                                 ;#1
   stream_append $dispatcher $input    ;#2
```

D. Input handler

```
stream_create M
set consumers [list S1 S2 S3]
register_agent M $consumers {
     set  consumers $argv
     foreach consumer $consumers {
          set counter($consumer)   0                ;#1
          stream_open $consumer                     ;#2
     }
     set count 0
     on _stream_append  {
          incr count
          set loc [stream_sender_name]              ;#3
          incr counter($loc)
          foreach stream $consumers {
             stream_append $stream $argv            ;#4
          }
          if { $count == 50 } {
             # find_max finds the max counter value
             set newloc [find_max counter]
             stream_relocate [stream_location $newloc]   ;#5
          }
     }
      on_stream_relocation {
          foreach consumer $consumers
             set counter($consumer) 0                   ;#6
          }
          set count 0
      }
}
```

B. Mobile Dispatcher

Figure 6. A simple auto-reconfiguring distributed chat that uses mobility to minimize latency for the interactive user. See Section 4.1.

called *TKShare* for sharing unmodified TK applications using the "What You See is What I See" (*WYSIWIS*) paradigm. Each user runs a separate copy of the application. What each user inputs to her GUI must be replayed on every other user's copy of the application so that the *WYSIWIS* guarantee may be preserved. We call these actions GUI events. The application may be sensitive to the order of input actions and hence, the input GUI events must be replayed in the same order on each user's copy of the application.

TkShare works by re-binding each TK widget based on the approach of *TkReplay* [4]. The re-binding code visits all the widgets in the widget hierarchy and finds the tags bound to it. Then it finds each GUI event callback that is bound to the tag and re-binds it. For each binding, the original script is saved in a table and replaced by a binding that sends the action to a central GUI event re-dispatcher *MStream* using an *append* in much the same way as the network chat example, bypassing the binding script that would normally get called. The re-dispatcher *MStream* resends the GUI event to each participant by *appending* to a stationary *MStream* located on each participant's machine. On reception of the *append*, at this stationary *MStream*, the original binding script is invoked.

Following the same strategy as the network chat example, the central mobile dispatcher periodically repositions itself to minimize latency for the currently most interactive user. This example points the way to how mobility can be used to achieve multi-party low sender-latency totally ordered reliable multicast service without the need for global locking.

4.2 Distributed Test Scripting

Web-based distributed testing has many applications such as protocol inter-operability testing and peer-to-peer performance testing. In this application, we use AGNI as a runtime Middleware for a high-level distributed test scripting application. We motivate this application with an example.

To test the throughput over a wide-area connection, a reasonable test sequence might consist of the following actions: (1) Set up a receiver process on a site; (2) Set up a sender process on another (geographically separated) site; (3) Connect from the sender to the receiver process; (4) Transfer an amount of data from the sender to the receiver; (5) Report the timing back to some central test collection *Site*.

If we were working in a closed system where failures are not possible and all resources are free to be used by the test, such a test scenario would be trivial to construct. However, consider performing the same test over the public Internet. Here, we wish to sample the bandwidth between widely-dispersed users who agree to run the test. Clearly, for the simple test described above, at least two users need to be present at any time and both must agree to exchange data. Following are are design requirements for the test system:

Test Coordination: In general, there is a precondition that needs to be established before the test can begin its execution, and in more complicated scenarios some intermediate coordination between test sites that needs to be done as the test progresses.

Test Structure and Deployment: The logical structure of the distributed test application may be known *a priori* but the physical structure depends upon the IP addresses of the physical machines involved. We thus need a means of dynamically mapping a logical distributed application structure to physical machines.

Failure Handling: User sites may disconnect before the test completes. In this case the test results may not be considered accurate. We need a means to stop the test and log the failure under these circumstances. Alternatively, if the test is a long-running test of a distributed system, we may wish to reconfigure the test dynamically while it is in execution.

Security: Users may not be trusted, and further, may not trust each other. The test system must thus provide users the ability to authenticate each other and to specify local policies that protect their own resources.

Actions taken by each of the participating workstations is triggered by discrete changes in state that drive the distributed system into new states, thus making the test driver event-driven.

While the requirements of such a system may be met by directly scripting in AGNI, users may wish to use higher level constructs to describe the parallelism and concurrency of such a test. We implemented a Simple Task Graph Language (STGL) to meet the requirements above. STGL has very simple constructs to describe concurrency, parallelism and distribution. For example, a simple test specification in STGL may look as specified in Figure 8(A).

The TCL code in Figure 8(A) is largely self-explanatory. A distributed task is specified as a set of subtasks with dependencies. Each subtask section can have a list of inputs and is demarcated by %{ and %}. Tasks can communicate with each other via a mailbox structure that is implemented in AGNI. A task may place intermediate results in a mailbox where other tasks can pick them up. A task blocks if it attempts to read a mailbox which is empty and is unblocked when a result arrives.

There is a distinguished mailbox for the result of the task graph which is set when the task graph completes execution or fails to which a result can be sent via the *set_result* function. In the example above, the function *ask_user_permission* puts a pop-up in front of the participant and asks permission to run the test on her machine. *start_receiver* sets up the receiver and *start_sender* sets up a process that does an active connection to the receiver end and sends the number of data exchanges required. The result is reported by posting to a *collector mailbox*. Finally, we clean up all the intermediate structures needed to manage the test.

The *MStreams* mechanisms presented in the previous sections are used to instantiate and run the distributed test-scripting application as described in the next section.

4.2.1 The runtime system for STGL

The basic template for each task is instantiated with two *MStreams* — one to enable the task and another to which the task posts a result with the body of the task appearing as an *append Handler*. When a message arrives at the *MStream* the *append* handler is executed and the data is consumed. After the task has executed, the runtime system posts a message to the enabling *MStream* of all dependent tasks.

The *Mobile Streams* are placed at the appropriate *Sites* to run the task graph. *Appends* to the *Streams* are transitions that result in execution of the action parts.

Figure 8(B) depicts the AGNI intermediate code that is generated to implement the task graph in 8(A). Each distinct task is executed through the *action* subroutine

```
set place0 0
set place1 1
set place2 2
set npings 100

Taskgraph mytask -task {
    sequential begin
    # sequential section.
        concurrent begin
        # concurrent section.
            at $place1 inputs: $npings %{
                # Ask user permission to execute test
                set result [ask_user_permission ]
                if { $result == "no" } {
                    set_result "fail"
                }
            %}
            at $place2 inputs: $npings %{
                set result ask_user_permission
                if { $result == "no" } {
                    set_result "fail"
                    return
                }
                start_receiver $npings
            %}
        end

        at $place1 inputs: $place2 $npings%{
            set ip_addr [get_peer_addr $place2]
            if { [connect_to_receiver $ip_addr] == 0 } {
                set_result "fail"
            }
            set start_time [clock_clicks]
            send_data $npings
            set delta expr [ [clock_clicks] - $start_time ]
            set_result "pass $delta"
        %}

        concurrent begin
            at $place1 %{
                puts "Thank you for participating"
            %}
            at $place2 %{
                puts "Thank you for participating"
            %}
        end
    end
}

set result [mytask.run]
if { [lindex $result 0] == "fail" } {
    puts "Test failed or aborted"
} else {
    set ping_time [lindex $result 1]
    puts stdout "Result = $ping_time"
}
$temp_streams [mytask set temps]
foreach i [$temp_streams] {
    stream_destroy $i
}
```

(A) Test Script Specification

```
action 1 {} {
    set result [ask_user_permission]
    if { $result == "no" } {
        set_result "fail"
    }
} Start2 t0

action 2 {npings 100} {
    set result [ask_user_permission]
    if { $result == "no" } {
        set_result "fail"
       return
     }
     start_receiver $npings
} Start2 t0

action 1 {place1 1 npings 100} {
    set ip_addr [get_peer_address $place1]
    if { [connect_to_receiver $place1] == 0 } {
         set_result "fail"
         return
    }
    set start_time [clock_clicks]
    send_data $npings
    set elapsed_time [expr [clock_clicks] - $start_time]
    set_result "pass $elapsed_time"
} t0 t1

action 1 {} {
     puts "Thanks for participating"
} t1 Done3

action 2 {} {
     puts "Thanks for participating"
} t1 Done3
```

(B) Generated Intermediate Code

```
proc action { loc input action pre post } {
  set astream [stream_create /webtest -temp]
  stream_move $astream $loc
  register_agent $astream  \
    [list $loc $input $action $post] {
    extract_args {loc input action post} $argv
    on_stream_append {
        if {[catch {eval $action} result] == 1} {
           report_error $result
        }else {
            stream_append $post "ACTION_DONE"
        }
     }
     on_stream_failure {
        stream_append "COLLECTOR" "FAIL"
     }
   }
   # Create and attach pre/postcondition handlers
   stream_create /webtest/$pre
   attach_precondition_handlers
   stream_create /webtest/$post
   attach_postcondition_handlers
}
```

(C) Partial listing of "action" function

Figure 8. A Simple Task Graph Language (STGL) for test scripting. Illustrates the use of a mobile-stream based run-time system for high-level test scripting. The script is specified as concurrent and sequential sections as shown in (A) and compiles to the intermediate code representation in (B). Mobility is used for instantiation of the system. Error recovery is effected by on_failure processing.

(see figure 8(C)) that takes 5 arguments. The first is a *Site* location identifier, the second some initialization code, the third is the body of the subtask, the fourth an *MStream* (called the trigger *MStream*) which triggers task execution an *MStream* name to which an *append* message will be sent after the task has completed execution. Note that action creates temporary *MStreams* for the transitions as stream names are not relevant so long as they are distinct.

The *run* method of the Taskgraph (invoked in Figure 8(A)) instantiates the system by compiling the task graph and generating the code shown in Figure 8(B) which is then sourced. The *action* subroutine creates the appropriate MStreams, moves them to the target *Site* and

registers agents that in turn register *append* and failure handlers.

The *on_stream_append* handler for each trigger *MStream* is the subtask code supplied to the *action* subroutine. Our failure assumptions are simple – on failure we just want to abort the test and note the failure. Hence, each *MStream* has the same *on_failure* handler that simply *Appends* a "FAIL" message to the *COLLECTOR MStream* .

The COLLECTOR is a stationary *MStream* residing in the reliable *Session Leader Site* that has a handler that cleans up all intermediate streams and aborts the test.

In order to have the system *auto instantiate*, we define an *on_new_peer Handler* in the *Session Leader*. The *on_new_peer Handler* is invoked for authorizing new connections. We use it to set up the test system by moving the intermediate *MStreams* to the participating workstations when they connect to the server.

5 Related Work

In contrast to other research in Mobile Agents, our approach has been to treat mobility and Mobile Agent technology as an enhancement to distributed scripting rather than as a means of supporting disconnected operations. Consequently, we have concentrated on typical distributed systems issues such as location tracking, message passing and failure handling. This distinguishes and separates our work from the other work in this area.

In this work, we proposed direct communication (reliable message passing) for communication between Mobile Agents. In contrast to mobile TCP [1], what we are really trying to implement is communication "stack" mobility; whereas, in mobile TCP, the entire machine (including its communication stack and all its state) moves from cell to cell.

In our system *on_stream_append* Handlers (analogous to "Agents" in other systems) pass one-way messages to each other reliably (via *MStreams*) rather than meeting to exchange messages, or using RPC-like mechanisms. Cabri et. al. [2] state that direct co-ordination or message passing between Agents is not advisable for the following reasons: (1) the communicating Agents need to know about each others existence (2) routing schemes may be complex and result in residual information at the visited nodes and (3) if agents need to communicate frequently they should be co-located anyway. They suggest black-boarding style of communication for Mobile Agents. They also note that direct message passing has the advantage of being efficient and light-weight. We concur with their concerns for the case of free-roaming, disconnected agents without any point of control. However, our system is oriented towards building re-configurable distributed applications where the logical structure of the application is known a-priori with the *Session Leader* having overall knowledge of the location of each *MStream* , thereby alleviating concern (1). Second, our communication protocol relies on sender initiated re-transmission, rather than forwarding, thereby eliminating the concern over residual information. All the necessary state information is restricted to the *Session Leader* and the sender. Third, our system model and message passing mechanisms do not preclude co-locating agents that communicate frequently - as illustrated by the auto-reconfiguring chat example. The application can use the same message passing primitives regardless of the position of the target *MStream* with which it is attempting to communicate. Finally, we note that a black-boarding scheme can be built on top of the mechanisms that we provide using synchronous (round-trip) *Append* messages and MStream blocking that are not described in this paper for reasons of brevity.

Our framework and toolkit is related to several other systems that support mobility. Systems such as Agent Tcl [6] and ARA [10] support a generalized mobility model where migration is allowed at arbitrary points in execution of the mobile code. Previously, we had developed a system called *Sumatra* that supports unrestricted mobility for Java applications by modification of the Java Virtual Machine [12]. Unrestricted mobility makes support of fault tolerance and reconfiguration harder to achieve. In contrast, our system restricts mobility and other state changes to handler boundaries and treats handlers as atomic. By providing such a clean execution model, we simplify both the system design as well as the design of applications built on top of our system.

Systems such as Aglets [9], Voyager [3], TACOMA [8] and Mole [13] follow a programming model similar to ours. However, our system differs from these systems in the following important ways: (1) Our design philosophy is to incorporate reconfiguration into a distributed system building toolkit rather than support disconnected operation as the fundamental design goal, (2) We have incorporated a peer-to-peer reliable, resilient message delivery protocol that none of these other systems offer and (3) We have a means of restricting system re-configuration and extension using policy *Handlers* that separate global (system-wide), and local concerns.

Dynamic re-configuration of distributed systems has been considered by Hofmeister and Purtilo [7] using a

software bus approach. Their system supports dynamic changes to modules, geometry and structure of a distributed system. However, failure processing and asynchronous message delivery during reconfiguration is not considered.

Our work relates to the work in mobile object systems - notably the *Globe* system [14]; however, our design goals are significantly different than those of *Globe*. Our proposed system could be used as an underlying infrastructure for a distributed mobile object system as our design is at a lower level than objects. *Globe*'s location tracking hierarchy offers great scalability but could result in higher latency for object location. Our focus is on providing an infrastructure for distributed control and hence we envision less stringent need for scalability but greater emphasis on latency reduction.

Our design is related to the work on the META toolkit [15]. The key difference is that we incorporate dynamic reconfiguration. META uses causal communication primitives offered by ISIS and we do not. However, it may be noted that with FIFO ordering and *Mobile Streams* the same effects can be achieved as with causal primitives. We keep time-stamp vectors to ensure FIFO message consumption in the presence of *MStream* motion but do not pay the overhead of transmitting entire timestamp vectors when the streams are not in motion.

6 Conclusions and Future Work

In this paper we have presented the motivation and design of a Middleware framework that enables the use of mobile agents to simplify distributed scripting. In particular we found that restricting reconfiguration and system extension to handler boundaries simplifies the implementation of the system while imposing some burden on the programmer to design the mobile application with these constraints.

Our plans for extending the Middleware is concentrated in two areas. We will incorporate reliable multicast primitives in our system whereby an *MStream* can communicate with a group of *MStreams*. As in the unicast case, both the sender and the recipients can be in motion while messages are being delivered. Our work will build on the earlier work in scalable, reliable multicast (SRM) [11]. Our justification for using a scheme such as *SRM* stems from its inherent scalability. Further, SRM is well suited for small, bursty messages. The message traffic in distributed control systems is likely to be bursty and sporadic.

Second, we intend to make our location tracking scheme more robust and scalable by using replication and multicast.

We intend to continue building applications - especially in the domain of mobile computing and distributed testing. We are currently building a data-management application that manages data collection and visualization in a distributed, "collaboratory".

7 Acknowledgments

The authors acknowledge and appreciate the contributions of Laurent Andrey (LORIA, Fr.) and Anurag Acharya (UCSB) both of whom contributed ideas to the design of AGNI and to Fernand Pors for using AGNI to build applications. Marc Bednarek (NIST) performed the performance tests for the results presented in this paper. We thank Kevin Mills, Mark Carson and Craig Hunt of NIST, the anonymous referees and our Shepard for reading this paper and making useful suggestions to improve its content, readability and presentation. This work was supported in part by DARPA under the *Autonomous Negotiation Teams* (ANTS) program (AO # 99-H412/00).

References

[1] H. Balakrishnan, V. Padmanabhan, S. Seshan, and R. Katz. A comparison of mechanisms for improving tcp performance over wireless links. In *ACM SIGCOMM*, August 1996.

[2] G. Cabri, L. Leornardi, and F. Zambonelli. Coordination in mobile agent systems. Technical Report DSI-97-24, Universita' di Modena, October 1997.

[3] Object Space Corp. Voyager white paper. http://www.objectspace.com/voyager.

[4] Charles Crowley. Tk-replay : Record and replay in tk. In *USENIX Third Annual Tcl/Tk Workshop*, 1995.

[5] Jonathan Dale. *A Mobile Agent Architecture for Distributed Information Management*. PhD thesis, University of Southampton, September 1997.

[6] Robert S. Gray. Agent Tcl: A flexible and secure mobile-agent system. In *Proceedings of the Fourth Annual Tcl/Tk Workshop - Monterey CA*, July 1996. http://www.cs.dartmouth.edu/ agent/papers.html.

[7] Christine R. Hofmeister and James M. Purtilo. Dynamic reconfiguration of distributed programs.

In *11th. International Conference on Distributed Computing Systems*, pages 560–571, 1991.

[8] Dag Johansen, Robbert van Renesse, and Fred B.Schnieder. An introduction to the TACOMA distributed system. Technical Report 95-23, University of Tromso, Norway, June 1995. *http://www.cs.uit.no/DOS/Tacoma/tacoma.webpages*.

[9] Danny B. Lange and Mitsuru Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998. ISBN 0-201-32582-9.

[10] H. Peine. *in Mobile Agents*. Manning Publishers, 1998. ISBN 1-884777-36-8.

[11] Suchitra Raman. Design and analysis of a framework for reliable multicast. Master's thesis, Dept. of Computer Science, Univ. of California, Berkeley, CA, May 1998.

[12] M. Ranganathan, Anurag Acharya, Shamik Sharma, and Joel Saltz. Network-aware mobile programs. In *USENIX Winter Technical Conference*, jan 1997.

[13] Markus Straer, Joachim Baumann, and Fritz Hohl. Mole – a Java based mobile agent system. In *2nd ECOOP Workshop on Mobile Object Systems*, pages 28–35, Linz, Austria, July 1996. http://www.informatik.uni-stuttgart.de/ipvr/vs/Publications/1996-strasser-01.ps.gz.

[14] M. van Steen, P. Homburg, and A.S. Tanenbaum. The architectural design of globe: A wide-area distributed system. Technical Report IR-422, Vrije University, March 1997.

[15] M. Wood and K. Marzullo. *in Reliable Distributed Computing with the ISIS toolkit*. IEEE Computer Society Press, 1994. ISBN 0818653425.